13 — Interfaces

🕒 state of the art 2026-07 · revised 2026-07-26 · 📖 ~12 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Argue** why "core with front-ends" beats "front-end with an agent inside" — and how drawing the boundary early maximizes the possible surfaces;
2. **Distinguish** the surfaces (TUI (Terminal User Interface), headless/SDK (Software Development Kit), IDE (Integrated Development Environment), chat, cloud) and what each one demands from the core;
3. **Evaluate** the interaction UX in light of HCI (Human-Computer Interaction) (mixed-initiative, levels of automation, over-reliance);
4. **Recognize** the surface as a security boundary (same turn contract, not a backdoor) and the shift to the *ambient*/inbox paradigm;
5. **Explain** why, with the loop behind ports, a second surface (headless) is a thin adapter, not a rewrite (step 0 of harness-zero).

The problem

The same agent must serve different audiences: the developer at the terminal, the CI script that needs JSON, the IDE that wants inline diffs, the manager who follows along over chat. The architectural question is a single one: **is the harness a core with multiple front-ends, or a front-end with an agent inside?** The harnesses we studied answered "core with front-ends" — and the quality of that separation determines how many interfaces are viable.

Established surfaces: **interactive TUI**, **headless/non-interactive** (`-p` with structured output), **IDE** (diffs, editor context), **CI/CD** (Actions), **agent protocols** (ACP (Agent Client Protocol), A2A (Agent-to-Agent)), **chat** (Slack, Telegram…) and, increasingly, **cloud/asynchronous**.

Scientific foundations

An honest editorial note (Principle I): **there is no academic canon of "agent harness interface"** — the gap is real. But the HCI of human-AI interaction grounds it with precision, and a recent trickle (2025-26) already addresses human-in-the-loop for agents.

- **When to act × when to ask** — Principles of Mixed-Initiative UI (Horvitz, CHI '99): the 12 principles about goal uncertainty, the cost/benefit of acting, and graceful handoff *are* the central decision of a harness — plan mode and approvals (chs. 07/09) are "passing the initiative", applied.

- **The autonomy dial is per stage** — the 10-level automation scale (Sheridan & Verplank, 1978) and the types-and-levels model (Parasuraman, Sheridan, Wickens, 2000) show that automation applies *independently* to each stage (acquisition · analysis · decision · action). Decision: the harness can **auto-collect context** (high automation) and still **gate the action** (low automation) — the dial does not have to be global.
- **The UX of "when it errs"** — Guidelines for Human-AI Interaction (Amershi et al., CHI '19): 18 guidelines by phase; the recovery ones (cheap correction/undo) explain why reversibility (ch. 08) is also an *interface* decision.
- **Oversight is fragile — design against that** — To Trust or to Think (Buçinca et al., CSCW '21) shows that explanation alone does **not** cure over-reliance; cognitive *forcing functions* do. The over-reliance review (Passi & Vorvoreanu, MSR-TR-2022-12) synthesizes the risk. Decision: approval must be a **deliberate act**, not a reflex click, and the surface cannot hide what the agent did. Recent human-in-the-loop agent work (Magentic-UI, arXiv 2507.22358, with *action guards* = permission gating; oversight design, arXiv 2510.19512) operationalizes this.

(Full bibliography and pointers: `livro/bibliografia.md`.)

Industry sources

- **One core, many surfaces (now doctrine)** — the Platforms and integrations (Claude Code) doc says it explicitly: "runs the same underlying engine everywhere, but each surface is tuned to a way of working" (CLI, Desktop, VS Code, JetBrains, Web, Mobile + Chrome, GitHub Actions, GitLab, Slack), with **config, project memory and MCP (Model Context Protocol) shared** across the local surfaces. Decision: build the agent as a single engine and treat terminal/IDE/web/mobile as interchangeable front-ends.
- **Headless is a Unix filter** — Run Claude Code programmatically (headless): `-p`/`--print`, `--output-format text|json|stream-json`, reads stdin and redirects stdout "like any command-line tool", with `--allowedTools`/`--permission-mode` so unattended runs never hang on a prompt. Decision: the interface is stdin/stdout + exit codes — the agent drops into pipes, build scripts and CI without a UI.
- **The SDK is the loop packaged; Managed Agents is the agent as a service** — the Agent SDK provides "the same tools, loop and context management that power Claude Code", programmable in Python/TS, and separates *who runs the loop* (SDK in your process) from *who renders it*; Managed Agents take it to the extreme — "Anthropic runs the agent and the sandbox, your application sends events and receives the stream". Decision: the programmatic surface is the core as a library — or as a REST endpoint.
- **The IDE is a thin surface over the same engine** — VS Code and JetBrains add inline diffs and editor context by reusing the CLI engine (same CLAUDE.md, same permission modes); the broader pattern — Copilot agent mode, Cursor 2.0 (background agents), Windsurf Cascade — splits the editor surface into **inline (synchronous)** and **background (asynchronous, cloud)** over the same task abstraction.
- **The interaction UX: approval as a state machine, streaming as events, the human as a tool** — the permission modes turn approval into a state machine (default/acceptEdits/plan/…), not an ad-hoc prompt; streaming exposes the loop as a typed event stream (`text_delta`, `tool_use`, `result`); and

AskUserQuestion models human-in-the-loop **as a tool** the agent calls — "asking the human" becomes a step of the loop, not a special interruption.

- **The ambient/inbox shift** — for asynchronous agents, the surface stops being the chat prompt and becomes an **inbox**: LangChain's ambient agents (always-on, event-triggered, surfacing to the human only via notify/question/review) and Claude Code on the web (runs in a managed cloud and "keeps going after you disconnect") point to the same future: supervising *many* long-running agents without a live terminal — exactly the "oversight without constant oversight" that levels-of-automation HCI predicts.
- **Chat as a service, with its own identity** — channels let Telegram/Discord "or your own server" push events into a session; Slack turns `@Claude` into a cloud session that transforms a bug into a PR, **with its own credentials and audit trail, decoupled from any human's access**. Decision: chat is just another trigger, and the agent-as-service has its own identity (ties into ch. 07).
- **See also**: the living collection Awesome Harness Engineering — Human-in-the-Loop gathers more consultable resources for this dimension (patterns, articles and implementations), curated by problem.

The state of the art

1. Core with front-ends — the early boundary decides everything

The structural lesson of round 1 has become consensus: **the earlier the core/interface boundary is drawn, the more interfaces fit later**. Codex is the crystal-clear example — "a single Rust engine serves the TUI, headless `codex exec`, the IDE extension, the desktop app, cloud/web, an MCP server and remote control". opencode pays for the same bet with a typed HTTP API and generated clients. The anti-pattern is the inverse: a front-end with an agent stuffed inside, which does not scale to a second surface without a rewrite.

2. Headless with structured output is mandatory

There is no serious harness without the Unix-filter mode: `codex exec` (JSONL), gemini `--output-format stream-json` (NDJSON of events), `oh -p`/`ohmo --print`, Aider headless. Structured output is what makes the agent **programmable** — a pipeline piece, a CI target, the backend of another UI. It is the surface that, once absent, closes off every other automation.

3. Three visions — and the explosion of the "colleague in chat" with voice

The three bets of round 1 persist, now sharper: the agent as a multi-platform **product** (opencode Electron/VS Code; Codex desktop+cloud), as a platform **service** (gemini-cli SDK/A2A/Action; Managed Agents REST) and as a **colleague** in chat. The personal-agent category has *exploded* the third one: **OpenClaw** serves **~23 chat channels** + native apps (iOS/Android/macOS/Windows) + **voice** (Voice Wake, continuous Talk Mode) + Live Canvas; **Hermes** has a single-process multi-channel gateway (10 platforms + voice); **ohmo** makes Telegram/Slack/Discord/Feishu the primary surface. Voice and channel breadth have become first-class surfaces.

4. The surface is a security boundary, not a backdoor

The most mature lesson of round 2, and the one that over-reliance HCI reinforces: a surface cannot be a shortcut around the core. **IronClaw** makes this concrete — CLI, WebUI, Slack, Telegram and webhooks all enter through the **same turn contract** (`ProductAdapter`), and the WebUI is *forbidden* from bypassing the auth boundaries. The agent-in-Slack runs with its own credentials and audit trail, decoupled from the human's. And the UX must not *hide* what the agent did — the antidote to the false sense of oversight that Buçinca and Passi & Vorvoreanu document.

5. The next frontier: ambient, cloud, asynchronous

The emerging paradigm changes the very nature of the interface. **Codex cloud-tasks** (a TUI for remote tasks), Claude Code on the web continuing after you disconnect, and the ambient agents' inbox all point to the same place: the human stops *driving* one live agent and starts *supervising many* through notification and review. It is HCI's autonomy dial taken to product — high automation in execution, the human at the decision gate, asynchronous. The agent's interface is leaving the terminal (Aider's watch mode turns `ai!` comments in any editor into commands; OpenClaw's Live Canvas) and becoming an environment.

> **EXECUTIVE SUMMARY**
>
> What is most modern: one engine, many surfaces (doctrine); structured headless as mandatory; the channel + voice explosion; the surface as a security boundary (same turn contract); and the ambient/inbox/cloud shift. **What to steal:** draw the core/interface boundary early (typed API or library), not late; ship headless with `stream-json` from day one; make every surface pass through the same turn contract (never an auth backdoor); model human-in-the-loop as a tool and approval as a deliberate act; and get ready for the inbox — the next terminal is asynchronous.

Hands-on — harness-zero: the chat as observation window

The harness-zero interface was born in **step 0**: a minimal chat on FastAPI, the *observation window* that accompanies every step of the book. The lesson of this dimension is what the entire project demonstrates: because the loop lives behind ports (`LLMPort`, `ToolPort`, `StorePort`), adding a **second surface** — a headless `--print` mode that emits the same events in `stream-json` — is a **thin adapter**, not a rewrite. Completeness exercise: you add the headless mode and prove that the same agent responds in the chat and in the pipe, and that approval (the permission gate from ch. 07) shows up on both surfaces through the same contract — the surface is not a backdoor.

Check your understanding

1. Why does "core with front-ends" allow more interfaces than "front-end with an agent inside", and what decides this in practice? (The boundary drawn early — typed API/library — lets each surface be a thin adapter; the inverse demands a rewrite per surface.)

2. Your agent will run asynchronously in the cloud, supervised by several humans. Which interface paradigm and which HCI principle guide the design? (Ambient/inbox — notify/question/review; levels of automation: high in execution, human at the decision gate; over-reliance → do not hide what the agent did.)
3. You expose the agent on Slack and in a WebUI. What rule prevents the new surface from becoming a security hole? (Same turn contract/ProductAdapter — every surface passes through the same auth boundaries; the UI does not bypass them; its own identity/audit trail.)

Appendix A — How each repository handles interfaces

> Evidence per harness, with paths — online supplement, expanded each round.

opencode (round 1) — the largest product surface

Client-server architecture (ch. 02): an HTTP server with a typed API and generated clients enables **seven surfaces** — TUI (SolidJS/opentui), **Electron desktop app** (unique in round 1), VS Code extension, **GitHub Action** (`packages/github/`), **Slack** (`packages/slack/`), web (`packages/web/`) and **ACP** (Zed integration). Link-shareable sessions connect the surfaces.

gemini-cli (round 1) — rich terminal + platform

React/Ink TUI with ~40 slash commands via pluggable loaders. **First-class headless**: `gemini -p` with `--output-format stream-json` (real-time NDJSON). **VS Code companion** (an IDE server exposing files/diffs). Official GitHub Action. **ACP** for editors and an **A2A server**. Its own SDK (`packages/sdk`).

OpenHarness/ohmo (round 1) — the agent that lives in chat

Typer CLI (`oh`) headless (`-p`, `text|json|stream-json`) + `--dry-run`; two TUIs (React/Ink + Textual); autopilot web dashboard. And **ohmo**: a personal agent on **Telegram/Slack/Discord/Feishu** (`channels/` + `gateway/`) with its own workspace.

OpenClaw (round 2) ⭐ — the widest surface breadth

~23 channels (WhatsApp, Telegram, Slack, Discord, Signal, iMessage, Teams, Matrix, Feishu, LINE, WeChat, QQ…), web Control UI, WebChat, CLI, TUI, **voice** (Voice Wake + continuous Talk Mode), **native apps** (iOS/Android/macOS/Windows) and **Live Canvas** (A2UI). The agent's surface as a consumer product.

Codex CLI (round 2) ⭐ — one engine, every surface

A single Rust engine serves: TUI (ratatui), `codex exec` headless (human + JSONL), IDE extension via App Server, **desktop app**, **cloud/web** (`cloud-tasks` with a TUI for remote tasks), Codex as an MCP server, and remote control. The canonical example of a core with front-ends.

IronClaw (round 2) ⭐ — same turn contract

CLI/REPL, WebUI (SSE+WS with OIDC, rate limiting, origin check), Slack, Telegram, webhooks — all entering through the **same turn contracts** (`ProductAdapter`); the WebUI is **forbidden from bypassing** the auth boundaries. The surface as a security boundary, not a backdoor.

Hermes (round 2) — single-process multi-channel gateway

Full TUI; **multi-channel gateway**: Telegram, Discord, Slack, WhatsApp, Signal, Email, iMessage, QQ, WeChat, Yuanbao — with cross-platform continuity; **voice** (multi-provider transcription + TTS); ACP for editors; an OpenAI-compatible API server.

Goose (round 2) — ACP desktop over an embedded core

Full CLI + TUI; **Electron desktop speaking ACP** to the core (embedded binary, no separate server); headless via recipes + scheduler; Telegram gateway and Discord bot; pure MCP/ACP server mode.

Aider (round 2) — input outside the terminal

Rich CLI/REPL (prompt_toolkit, streaming markdown), browser UI (Streamlit), **watch mode** (`aider/watch.py`: `ai!`/`ai?` comments in code from any IDE become commands), **voice-to-code**, images/URLs in chat. The interface escaping into third-party editors.

OpenHands (round 2) — SaaS control plane

React Web UI (~40 routes: conversations, settings, admin, billing, orgs); `agent-canvas` CLI; headless/REST via Agent Server; **GitHub/GitLab/Jira/Slack resolvers** (webhooks); full enterprise/SaaS (Keycloak, Stripe, multi-tenant); Docker/k8s deploy.

n8n (round 2) — embeddable chat + canvas

Chat Trigger (hosted chat app + embeddable `@n8n/chat` widget + streaming), Manual Chat Trigger, arbitrary webhooks, the visual editor (canvas) as the building interface, MCP Server Trigger. The "inverted harness" whose primary interface is the graph.

Frameworks (frameworks round)

The frameworks deliver the loop as a library (the pure programmatic surface) + event streaming + human-in-the-loop as composition (OpenAI Agents SDK, LangGraph, CrewAI); the UI is left to the integrator. It is the "core only, the surface is yours" extreme of the spectrum — the opposite of OpenClaw.